Un algorithme distribue pour l'execution parallele de PROLOG

F. Schwaab, D. Tusera

INRIA

UNITE DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
BP 105
78153 Le Chesnay Cedex
France
Tel. (1) 39 63 55 11

Rapports de Recherche

N° 935

Programme 2

UN ALGORITHME DISTRIBUE POUR L'EXECUTION PARALLELE DE PROLOG

François SCHWAAB
Dimitri TUSERA

Décembre 1988

UN ALGORITHME DISTRIBUE POUR L'EXECUTION PARALLELE DE PROLOG

A DISTRIBUTED ALGORITHM FOR PARALLEL EXECUTION OF PROLOG

François Schwaab * , Dimitri Tusera

INRIA
BP 105
78150 LE CHESNAY

* CRIN
BP 239
54506 VANDOEUVRE LES NANCY

Résumé :

L'apparition récente d'architectures de machines connexionnistes massivement parallèles, permet d'envisager l'exploitation du parallélisme inhérent à certaines classes d'algorithmes. En rapprochant les concepts de la lecture procédurale de Prolog, des réseaux sémantiques étendus de Kowalski et de la structure des machines connexionnistes, nous proposons l'implantation d'un algorithme d'interprétation distribué de Prolog mettant en œuvre le parallélisme intrinsèque du modèle de calcul de la programmation logique. Cet algorithme est implanté à l'aide de primitives traitant les messages circulant entre les processeurs, chaque processeur étant dédié au traitement d'un nœud du réseau et les messages se propageant selon le graphe définit par le réseau.

Abstract :

The recent development of massively parallel connexionnist architectures, suggests to use the implicit parallelism of certain classes of algorithms. Combining the concept of the procedural interpretation of Prolog, the extended semantic networks from Kowalsky and the connexionnist machine structure, we propose the implementation of a distributed interpretation algorithm of Prolog, using the implicit parallelism of the logical-programming computation-modell. This algorithm is implemented with primitives handling inter-processor messages : each processor is dedicated to a node of the semantic network and the messages are routed along the graph edges.

1. La forme clausale de la logique

Une **sentence** est une collection de **clauses** de la forme : A -> B_1,.....,B_n ; où A est une **conclusion** et B_1,.....B_n sont des **conditions**.
A, B_1,.....B_n sont des **prédicats** de la forme $P(t_1,.....t_k)$ où P est le nom du prédicat et t_1,.....t_k sont des termes. Les termes peuvent être des constantes, des variables ou des fonctions.
Pour une clause: A -> B_1,.....,B_n ; nous pouvons dire que A est vrai si B_1 et B_2 .....et B_n sont vrais.
Une **évidence** est une clause sans conditions: A -> ; qui est toujours vraie.
Une **contradiction** est une clause sans conclusion: -> B_1,.....,B_n ; qui est toujours fausse.
La **résolution** est une méthode de preuve de l'inconsistance d'une clause avec une sentence. Si nous voulons démontrer qu'une clause, nous l'appeleront le but, est déductible d'une sentence, nous prenons sa contradiction, c'est à dire seulement son côté gauche, et par les étapes successives de la résolution nous tentons de parvenir à une incohérence exprimée par une contradiction vide.

Voici un exemple:

Cherchons si le but : Aime(Jean,Marie)

est déductible de l'ensemble des clauses:

(1) Aime(x,y) ->Aime(x,z),Donne(x,z,y);
(2) Aime(Jean,livre) ->;
(3) Donne(Jean,livre,Marie) -> ;

Etape 1 :
La contradiction : -> Aime(Jean,Marie)

s'unifie avec la conclusion de la première clause (1)
et donne la contradiction: ->Aime(Jean,z),Donne(Jean,z,Marie)

Etape 2 :
la première condition s'unifie avec
la conclusion de la deuxième clause (2)
ce qui mène après unification de z et de livre
à la contradiction : -> Donne(Jean,livre,Marie)

Etape 3 :
celle-ci s'unifie avec la conclusion de la troisième clause qui mène à une contradiction vide.
Ainsi la consistance du but avec la sentence est démontrée.

2. L'Interprétation procédurale de PROLOG :

La clause A -> B_1,.....,B_n ; peut être lue comme une procédure ayant pour **tête** la conclusion A, et comme **corps** les conditions B_1,.....,B_n.
Ainsi une contradiction, le but, peut être interprétée comme une suite d'appels de procédures : chaque condition est un appel de procédure dont la définition formelle est donnée par la clause ayant la même tête.
Les principales différences avec les autres langages procéduraux sont les suivantes :

a/ une procédure peut avoir plusieurs définitions formelles,
b/ il n'y a pas substitution des paramètres de la procédure par les arguments de l'appel ; la liaison est effectuée par l'instanciation des arguments par l'unification.

Appelons les valeurs des arguments de l'appel: **Inputvals.** Elles peuvent être modifiées par l'unification et au retour de l'appel appelons les **outputvals.**

Exécuter un appel de procédure situé dans un but, signifie:

- instancier les inputvals de l'appel avec les paramètres de la tête de la procédure : les outputvals,
- remplacer les arguments dans le corps de la procédure appelée par ces nouvelles valeurs,
- substituer dans le but l'appel de la procédure par son corps,
- propager les outputvals dans tous les autres appels du but.

Ainsi nous obtenons un nouveau but qui est composé d'une suite de nouveaux appels, le corps de la procédure appelée, et de la suite des anciens appels. Pour poursuivre le traitement on peut exécuter soit les anciens appels dans le but soit les appels insérés récemment. L'interprétation standard de Prolog choisit systématiquement d'interpréter l'appel de procédure situé le plus à gauche dans la suite des appels insérés le plus récemment : stratégie "le plus à gauche et en profondeur d'abord".
Afin de permettre d'établir plus aisément une correspondance entre la syntaxe de Prolog et la structure d'une machine connexionniste, nous allons adopter une variante syntaxique particulière : les réseaux sémantiques étendus [1].

3. Représentation du programme par un réseau sémantique :

Chaque prédicat ayant n arguments, n > 2, peut être transformé en n+1 prédicats à deux arguments. Ainsi nous pouvons ne considérer que l'existence de prédicats à deux arguments et la représentation d'un programme Prolog par un réseau sémantique se réduit à une transcription syntaxique.

Voici les régles de transcription:
L'ensemble des clauses se transforme en un graphe orienté dans lequel :

- un nœud représente un argument de prédicat,
- un arc orienté représente un prédicat,
- l'orientation de l'arc reliant les deux nœuds exprime l'ordre d'écriture des arguments,
- les conclusions sont représentées par des flèches simples : -------->
- le conditions sont représentées par des flèches grasses : ==========>
- les arguments partagés par plusieurs prédicats sont représentés par des nœuds partagés

La figure 1 donne l'exemple d'une clause: grparent(x,z) -> parent(x,y),parent(y,z) ;

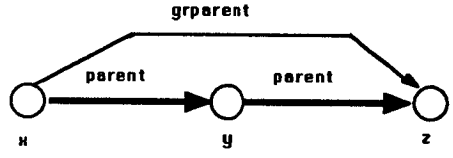

fig. 1

La figure 2 nous décrit un exemple d'interprétation :
le but :

grparent(Pierre,x)

est à résoudre dans l'ensemble des clauses:

(1) grparent(x,z) -> parent(x,y),parent(y,z)
(2) parent(Pierre,Paul) -> ;
(3) parent(Paul,Marie) -> ;

2.a/ situation de départ

2.b/ l'appel de la procédure grparent la fait remplacer par deux appels successifs de la procédure parent avec la variable **x** instanciée par la valeur **Pierre**. L'appel grparent(Pierre,x) et la conclusion grparent(x,z) disparaissent.

2.c/ l'appel de la procédure parent(Pierre,y) disparaît contre l'évidence parent(Pierre,Paul) -> ; et dans l'appel restant la variable **y** est instanciée à **Paul.**

2.d/ l'appel parent(Paul,z) revient à remplacer **z** par **Marie** ce qui mène à une contradiction vide.

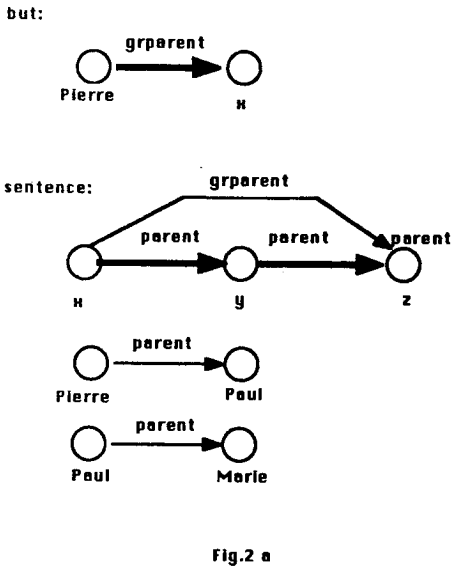

Fig.2 a

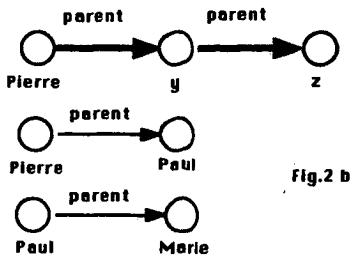

Fig.2 b

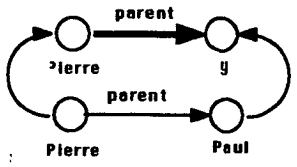

Fig.2 c

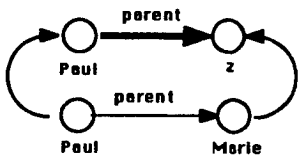

Fig.2 d

Le graphe du réseau sémantique met en évidence le parallélisme intrinsèque à l'éxécution du programme représenté. Avant de décrire l'algorithme d'interprétation distribué exploitant ces possibilités de traitement parallèle, nous rappelons les différents types de parallélismes inhérents au langage Prolog.

4. Parallélisme en Prolog :

Pour son exécution le langage Prolog offre trois types de parallélisme :

a/ pour obtenir toutes les solutions pour un but il faut pour chaque appel de procédure explorer toutes ses définitions formelles. Ces explorations sont mutuellement indépendantes et peuvent se faire en parallèle : **parallélisme OU.**

b/ il est difficile de lancer en parallèle les appels des différentes procédures dans la suite formée par le but car ils mèneraient chacun à différents ensembles d'arguments instanciés. Il faudrait ensuite effectuer l'intersection entre ces ensembles pour trouver les instanciations qui conviennent à tous les appels à la fois. La solution que nous proposons est d'exécuter la suite des appels du but d'une facon séquentielle. Ainsi l'appel suivant est activé dès que l'appel courant dégage une outputval : **parallélisme pipe-line ET.**

c/ au moment de l'appel d'une procédure on peut activer les deux nœuds de ses arguments : **parallélisme du traitement des arguments.**

5. Version simplifiée de l'algorithme distribué pour l'éxécution de Prolog :

Pour mettre en œuvre les trois types de parallélismes possibles dans l'interprétation de Prolog il parait naturel de projeter les nœuds du réseau représentant le programme sur un ensemble de processeurs et d'utiliser les connexions inter-processeurs pour implanter les liaisons du graphe [9].

Nous allons présenter successivement les structures de données et l'algorithme mettant en œuvre pour seul parallélisme celui du traitement des arguments.

5.1 Les connexions :

Les connexions inter-nœuds nécessaires à l'algorithme que nous proposons sont les suivantes :

1/ un nœud appartenant à un prédicat conclusion est connecté :

- à son partenaire dans le prédicat : lien N-p
- au nœud condition représentant l'appel de la première procédure (s'il ne s'agit pas d'une évidence) : lien N-first

2/ un nœud appartenant à un prédicat condition est connecté :

- à toutes les têtes des procédures synonymes : lien N-or
- au nœud de l'appel de procédure suivant s'il existe : lien N-seq
- au nœud conclusion s'il s'agit d'un nœud de la dernière condition d'une clause : lien N-c

La figure 3 illustre les connexions pour le réseau formé par le but : p1(x,y);
et l'ensemble des clauses : (1) p1(a,b)->p2(i,j),p3(k,l);
(2) p2(e,f)->;
(3) p2(g,h)->;
(4) p3(m,n)->;

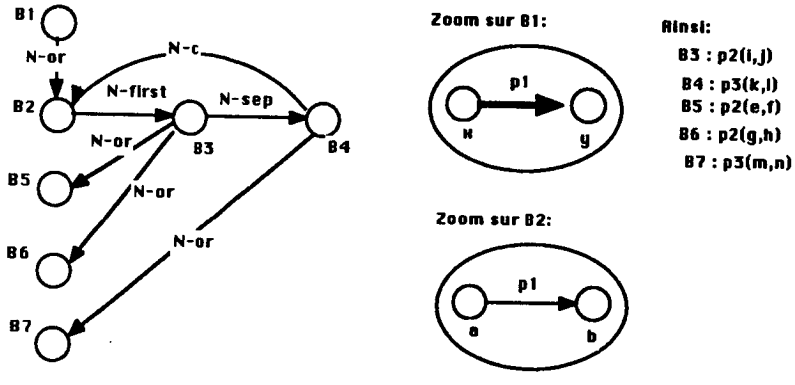

Fig. 3

Légende : nous représentons chaque prédicat par une boîte contenant les deux nœuds représentant les variables. Les prédicats sont reliés entre eux par des liens respectant les conventions définies ci-dessus, sachant que ces liens sont duppliqués pour les deux nœuds du prédicat.

Remarque : Cet exemple n'est guère réaliste car dans une sentence toute clause doit utiliser dans les prédicats conditions les paramètres du prédicat conclusion. Ainsi cet exemple est à l'image d'une procédure dont les instructions du corps n'utilisent pas les paramètres déclarés dans sa tête.

Dans notre modèle, l'utilisation d'un même objet, paramètre ou variable, par plusieurs prédicats d'une même clause s'effectue par le partage du nœud représentant cet objet. Ainsi un objet partagé évoluera en fonction de l'interprétation des prédicats auxquels il est attaché.

5.2 Les nœuds partagés :

Un nœud multiple est un nœud partagé entre plusieurs conditions et éventuellement avec la conclusion d'une clause. Dans l'exemple ci-dessous le nœud **b** est partagé entre la conclusion, prédicat **p1**, et la première condition, prédicat **p2**. Le nœud **c** est partagé entre les conditions **p2,p3** et **p4** (cf fig. 4a). Par conséquent le nœud **c** possède 3 listes de liens N-or pointant sur les en-têtes des procédures synonymes pour les prédicats **p2,p3** et **p4**. De même pour les connexions N-seq le nœud **c** possède 2 liens exprimant la séquence des prédicats **p2, p3** et **p3, p4**. Ces 2 liens pointent respectivement sur les prédicats **p3** et **p4** (cf fig. 4b).

Exemple :
```
(1)   p1(a,b)->p2(b,c),p3(c,d),p4(c,e);
(2)   p2(x,y)->;
(3)   p2(t,u)->;
(4)   p2(v,w)->;
(5)   p3(m,n)->;
(6)   p4(o,p)->;
```

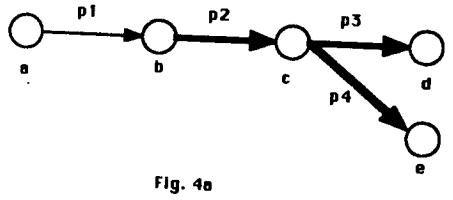

Fig. 4a

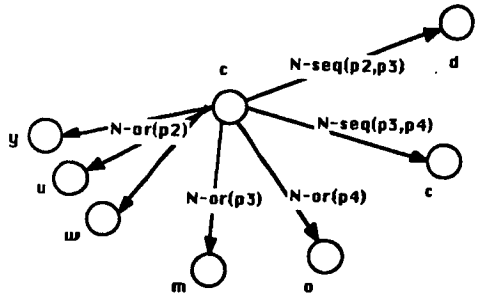

Fig. 4b

5.3 Algorithme simplifié :

Nous considérons dans ce paragraphe une interprétation strictement séquentielle de l'exemple de la figure 5 avec pour seul **parallélisme** celui du **traitement des arguments.** Chaque appel de procédure donnera lieu à l'activation en parallèle des deux nœuds des arguments de cette procédure. Après son éxécution, les variables instanciées sont transmises aux nœuds appelants.

Exemple : Soit le but : p1(x,y),p2(y,z);
et les clauses :
```
(1)  p1(a,b)->;
(2)  p2(c,d)->;
(3)  p2(e,f)->;
```

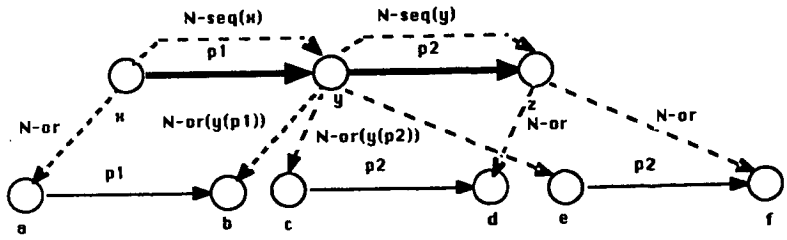

Fig. 5

Schématiquement les différentes étapes de l'interprétation seront les suivantes :

Appel de procédure pour **p1** :

1/ Les nœuds **x** et **y** activent les nœuds **a** et **b** en transmettant les inputvals,
2/ Les nœuds **a** et **b** réalisent l'unification entre les inputval et les valeurs locales et se synchronisent mutuellement sur le résultat communiqué : échec ou succès. S'il s'agit d'un échec, alors la branche de l'interprétation est abandonnée, sinon on continue. Considérons le cas d'un succès :
3/ Comme il s'agit d'une évidence, les nœuds **a** et **b** envoient les valeurs instanciées de l'appel aux nœuds appelants **x** et **y** : ce sont les outputvals de l'appel.
4/ En utilisant les liens N-seq, le nœud **y** active le nœud **z** et le nœud **x** active le nœud **y** pour la deuxième fois.

Appel de procédure pour **p2** :

5/ Le nœud **y**, en utilisant outputval du premier appel comme inputval de l'appel courant, et le nœud **z** appellent la première définition de la procédure **p2.**
6/ La suite de l'interprétation paraît évidente .....(cf paragraphe 2)

Remarque: le nœud **y** réalise l'appel de la procédure **p2** dès qu'il en a reçu l'ordre d'activation du nœud **x** et que inputval(p2), c'est à dire outputval(p1) lui est présenté : l'exécution se déroule selon un modèle dirigé par les données.

5.4 Les cinq primitives de l'interprétation :

L'algorithme d'interprétation sera distribué dans les nœuds (processeurs) contenant les variables. Son fonctionnement sera assuré par l'échange de messages entre ces nœuds, chaque message activant une primitive lors de sa réception par un nœud. Ainsi, un nœud activé par un message exécutera une fonction qui provoquera en général l'émission de nouveaux messages vers d'autres nœuds. Ce schéma d'exécution extrêmement simple permet de bénéficier du parallélisme intrinsèquement lié à la répartition des données dans les nœuds.

L'algorithme d'interprétation peut être exprimé à l'aide de cinq types de messages échangés entre les nœuds du graphe. A chaque type de message est associé une primitive exécutée lors de sa réception par un nœud.

5.4.1 msg(inputready, node)

Ce message informe le nœud désigné que inputval pour l'appel de procédure est actualisé.
Traitement :
- choix de la tête de procédure à appeler (parcours de la liste N-or),
- construction et émission du message msg(call, node N-or, inputval).

5.4.2 msg(call, node, val)

Ce message permet l'appel à une tête de procédure.
Traitement :
- unification entre la valeur du message et la valeur interne au nœud,
- émission du résultat de l'unification (succès ou échec) au nœud partenaire dans le prédicat par le message msg(synchro, node N-p, résultat : succès ou échec).

5.4.3 msg(synchro, node, résultat)

Ce message permet l'abandon de branches en échec.
Traitement : **SI** l'unification du nœud-partenaire et l'unification locale au nœud ont réussi :
alors si le prédicat est une évidence :
alors retour de outputval au nœud appellant,
construction et émission du message msg(return, N-caller, outputval);
sinon activation de la première condition de la partie droite par l'émission du message msg(inputready, node N-first);
sinon abandon de cette branche de l'interprétation.

5.4.4 msg(return, node, val)

Une procédure appellée rend la valeur instanciée d'un argument.
Traitement : **si** le nœud appellant fait partie de la dernière condition de la clause :
- **alors** la clause est vraie, tous ses nœuds sont instanciés, le nœud correspondant de l'en-tête est prévenu par le message msg(condOK,node N-c);
- **sinon** activation du nœud suivant dans la séquence des condition par l'émission du message msg(inputpret, node N-seq).

5.4.5 msg(condOK, node)

Un nœud conclusion est informé que toutes ses conditions ont été exécutées avec succès et que tous les nœuds contiennent des valeurs instanciées.
Traitement : le nœud conclusion peut signaler à son nœud appellant sa valeur instanciée de retour par msg(return, node N-appellant, outputval).

6. Structures de données de l'algorithme d'interprétation pour le parallélisme complet (OU,ET,arguments) :

En résumé, un programme Prolog est interprèté par l'échange de messages entre les nœuds du graphe, chaque nœud effectuant les actions suivantes :
- réception des messages,
- traitement sur ses données propres,
- émission de messages.

Pendant le parcours du graphe plusieurs problèmes dus au parallélisme se posent:

a/ Un appel de procédure peut activer plusieurs têtes de procédure. Lors des retours il faut que l'appellant puisse identifier de quel synonyme il s'agit.

b/ L'interprétation de l'appel d'une tête de procédure peut fournir plusieurs valeurs de retour. Il faut que les couples d'arguments instanciés par les valeurs de retour soient identifiés sans ambiguité.

Ces deux points sont résolus par l'association d'un identificateur **inputident** à inputval et d'un identificateur **outputident** à outputval. C'est le nœud conclusion qui pour chaque valeur de retour transforme inputident en outputident en le complétant par le numéro d'ordre de la clause identifiant le synonyme et le numéro du retour identifiant le couple de valeurs.

c/ Une tête de procédure peut à son tour être appellée plusieurs fois par la même ou par différentes conditions . Il faut, pendant l'exécution de son corps, marquer les différentes valeurs pour les associer aux appels.

Ce point est résolu en associant à chaque appel un identificateur unique : **callident**. En effet par suite de l'exécution pipe-line, les différentes parties du corps d'une procédure peuvent s'exécuter pour différents appels en même temps.

d/ Rappelons le problème des nœuds partagés : il faut associer à chaque inputval et à chaque outputval le prédicat concerné lorsqu'il s'agit d'un nœud partagé entre plusieurs prédicats.

Ce point est résolu en associant à chaque valeur : inputval ou outputval, le numéro du prédicat : **out**.

En résumé : Inputval est identifié par le numéro du prédicat : **out**, l'identificateur de l'appel : **callident** et l'identificateur associé à la valeur d'appel : **inputident**.
Outputval est identifié par le numéro du prédicat : **out**, l'identificateur de l'appel : **callident** et l'identificateur associé à la valeur de retour : **outputident**.

La suite de ce paragraphe et le paragraphe 7 présentent en détail les structures de données et l'algorithme complet d'interprétation. Cette présentation est nécessaire aux lecteurs désirant aborder l'implantation de notre algorithme. Nous invitons le lecteur pressé à nous rejoindre au paragraphe 8 traitant de la récursivité.

Pour présenter les données associées aux nœuds, nous les classons en deux catégories :
- les données statiques décrivant le graphe : attributs du nœud et liens entre les nœuds,
- les données dynamiques qui mémorisent les différents parcours du graphe : elles associent les variables aux différents messages circulant dans le graphe.

Dans notre description nous distinguons les nœuds-condition des nœuds-conclusion car de par leur rôle différent, ils manipulent des données différentes.

6.1 Les données décrivant le graphe :

Ces données sont obtenus lors de la traduction du programme Prolog en réseau sémantique étendu.

Pour les **nœuds-conditions:**

N-m : nom propre au nœud.
maval : valeur initiale de la variable ou de l'atome représentée par le nœud.
fct : nom de la fonction à appliquer éventuellement à la valeur d'un nœud. Pour un nœud partagé, on dispose d'un vecteur de noms de fonctions.
N-or : le vecteur des noms de nœuds conclusions synonymes du nœud condition avec un vecteur par prédicat pour les nœuds multiples.

N-seq : nom du nœud condition suivant à activer.
= nil s'il s'agit de la dernière condition d'une clause.
= un vecteur pour un nœud multiple avec un élément par prédicat.
Next : pour un nœud partagé next contient la liste des prédicats dans leur ordre d'enchaînement.
N-c : nom de nœud conclusion, il n'apparaît que dans la dernière condition d'une clause car une fois toutes les clauses satisfaites , il faut revenir à la conclusion.

Pour les **nœuds conclusions**

N-m : numéro d'ordre d'une tête de procédure dans un ensemble de synonymes. Il est nécessaire pour l'identification des outputvals lors du retour d'un appel.
maval : valeur initiale de la variable ou de l'atome représentée par le nœud.
m-p : nom de la fonction éventuellement à appliquer à la valeur du nœud-conclusion.
N-p : nom du nœud partenaire dans le prédicat.
N-first : le nom du premier nœud-condition à activer.

6.2 Les données décrivant le parcours du graphe:

Ces données mémorisent l'avancement de l'exécution du programme.

Pour les **nœuds-conditions :**

inputval et **outputval :** un nœud-condition simple appelle une procédure avec **inputval** et au retour reçoit **outputval**. Un nœud condition multiple utilise outputval du retour précédent comme inputval de l'appel courant ; s'il est partagé entre n prédicats, alors il possède n+1 valeurs. Le nombre de ces valeurs est connu statiquement.

out : quand un nœud active un autre nœud condition partagé, il faut qu'il lui indique pour quel prédicat il effectue cette activation. Ainsi out permet au nœud activé d'adresser les données : inputval, outputval, N-or, fct, N-seq.

Pour les **nœuds conclusions :**

tab : il s'agit d'une table où la conclusion mémorise tous les renseignements nécessaires au retour du résultat à un appel .

- les coordonnées de l'appellant:
  - son nom,
  - out dans le cas où l'appellant est un nœud partagé pour qu'il puisse rétablir les correspondances,
  - callident : identificateur de l'appel,
  - inputident : identificateur d'une valeur d'appel,
  - outputno : compteur de réponses envoyées permettant le calcul d'outputident.

- synchro : information permettant la synchronisation entre nœuds partenaires.
  = rien (l'unification n'a pas eu lieu),
  = succès (l'unification a lieu avec succès),
  = échec (de l'unification).

- inputval : résultat de l'unification.

Cet ensemble de valeurs est mémorisé pour chaque appel sous forme d'une ligne de **tab** . Le numéro de la ligne constitue le callident de l'appel.

synchro : l'activité des nœuds se propage dans le graphe d'une facon asynchronne selon un schéma dirigé par les données. Ainsi il peut arriver que l'activation des nœuds d'une conclusion soient complètement désynchronisée. Par exemple : un nœud est déjà sollicité pour plusieurs unifications tandis que son partenaire ne l'est pas encore. Dans ce cas le nœud en retard mémorise les demandes de synchronisation de son partenaire dans une liste **synchro** en attendant ses propres activations.

7. Algorithme d'interprétation :

Le traitement du parallélisme ET et OU n'augmente pas le nombre de messages différents échangés entre les nœuds. Ainsi cet algorithme est exprimé à l'aide de cinq primitives associées à la réception des messages. Remarquons que le premier argument du message est le nom du message ; le deuxième argument est le nom du nœud destinataire et les arguments suivants le contenu du message.

7.1 msg(inputready, node, out, callident, inputident)

Un nœud condition reçoit ce message lorsque sa valeur est actualisée et qu'il peut réaliser l'appel de toutes les procédures synonymes. Il leur transmet la valeur inputval retrouvée à l'aide de :

- out : identifiant le prédicat concerné par l'appel,
- callident : identifiant l'appel de la clause dans le corps de laquelle se trouve le nœud courant,
- inputident : identifiant inputval.

Traitement :

1/ Calcul de inputval:
SI le nœud condition est activé pour la première fois pour un callident donné
alors si le nœud est partagé avec la conclusion
alors inputval = résultat de l'unification effectuée lors du traitement du nœud conclusion;
sinon inputval = maval.
sinon inputval = outputval du prédicat précédent (*le nœud est partagé par plusieurs prédicats*)
SI le nœud possède une fonction **alors** inputval = fct(inputval).

2/ Appel des synonymes :
Pour chaque nœud appartenant à la liste N-or (*liste des synonymes du prédicat*)
émission du message msg(call, node N-or, inputval, out, callident, inputident, node N-m).
finpour.

7.2 msg(call, node, inputval, out, callident, inputident, node-return)

Ce message constitue l'appel à un nœud tête de procédure avec les paramètres suivants :
- inputval = valeur à unifier,
- out, callident, node-return = paramètres utilisés au retour,
- inputident = identificateur qui sera complété au retour pour former outputident.

Si lors de cet appel l'unification échoue, on abandonne le parcours de cette branche du graphe.

Traitement :

1/ Unification entre maval et inputval pour déterminer la valeur de outputval.
2/ **SI** l'unification a réussie
   alors result = succes
   sinon result = echec.
3/ Créer une nouvelle entrée dans tab de nom mycallident et y sauvegarder outputval, inputident, out, callident et node-return.
4/ Envoi du message msg(synchro, node N-p, mycallident, result : succes/echec, outputval).
5/ **SI** le message de synchronisation émis par le nœud partenaire est arrivé avec comme valeur succes **et** result = succes (* l'unification locale a réussie *)
   alors
      SI il s'agit d'une évidence
         alors envoi du message msg(return, node-return, outputval, out, callident, outputident)
         sinon out et inputident sont initialisés à nil (* il s'agit du premier appel à ce nœud *)
            envoi du message msg(inputready, node N-first, out, mycallident, inputident)
   sinon abandon de cette branche du graphe.

7.3 msg(synchro, node, callident, result, val)

Cette primitive permet de synchroniser le traitement associé aux deux nœuds d'une conclusion. On ne poursuit le parcours de cette branche du graphe que si l'unification du nœud partenaire et l'unification locale ont toutes les deux réussies.

Traitement :

SI l'unification locale n'a pas eu lieu
   alors mémoriser le message dans la liste synchro
   sinon si le message synchro a pour valeur succes et l'unification locale s'est terminée avec succès
      alors si il s'agit d'une évidence
         alors envoi du message
            msg(return, return-node, outputval, out, callident, outputident)
         sinon out et inputident sont initialisés à nil (* il s'agit du premier appel à ce nœud *)
            envoi du message msg(inputready, node N-first, out, mycallident, inputident)
      sinon abandon de cette branche du graphe.

7.4 msg(return, node, outputval, out, callident, outputident)

L'exécution d'une procédure rend un résultat au nœud appellant.

Traitement :

1/ Mémoriser outputval en utilisant les valeurs de out,callident et outputident
2/ Actualisation de out en utilisant la liste Next et inputident en utilisant outputident.
3/ **SI** il s'agit de la dernière condition d'une clause
   alors envoi du message msg(condOK, node N-c, out, callident, inputident)
   sinon envoi du message msg(inputready, node N-seq, out, callident,inputident)

7.5 msg(condOK, node, out, callident, inputident)

Le nœud-conclusion reçoit le message que la clause est satisfaite pour l'appel identifié par callident et inputident. Dans le cas d'un nœud partagé, outputval est retrouvé à l'aide de out.

Traitement :

1/ A l'aide des paramètres out, callident et inputident restaurer la valeur de outputval.
2/ A l'aide de callident restaurer depuis tab les paramètres de retour : out, callident et node-return.
3/ A l'aide de inputident, le numéro du résultat et le nom du nœud générer outputident.
4/ Envoi du message msg(return, node_return, outputval, out, callident, outputident).

8. La récursivité :

En Prolog séquentiel l'arrêt de la récursivite peut être explicite ou implicite.

L'arrêt explicite s'effectue de deux manières possibles:

a/ par une clause qui précéde la clause récursive et qui permet d'arrêter une branche de la récursivité
b/ à l'intérieur de la clause récursive on insère une condition d'arrêt avant l'appel récursif

L'arrêt implicite :

a/ l'unification échoue sur l'incompatibilité des paramètres

Dans la version de l'interpréteur Prolog distribué proposée nous respectons le fonctionnement de l'arrêt implicite. L'arrêt explicite nécessite quelques explications supplémentaires :

- l'ordre d'écriture des clauses n'a plus d'importance : la clause d'arrêt est exécutée en parallèle avec la clause récursive donc la condition a/ est vérifiée.
- la condition b/ est vérifiée si le séquencement du pipe-line ET des conditions est le même que dans un interprête Prolog séquentiel : d'abord la condition d'arrêt, ensuite l'appel récursif.

Nous invitons le lecteur à dérouler l'algorithme sur l'exemple suivant (fig. 6) :

Le but : membre(x,l);

La sentence : (1) membre(x,l) -> egal(x,car(l));
(2) membre(x,l) -> membre(x,cdr(l));
(3) egal(x,y) ->;

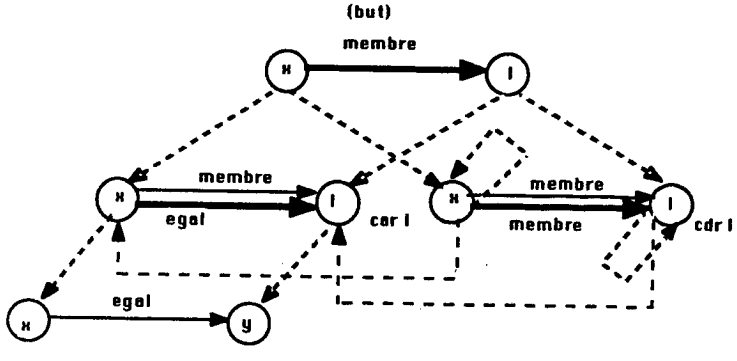

Fig. 6

9. Conclusion

L'algorithme présenté a été écrit en Lisp et sa validité a été testée sur un certain nombre d'exemples. Il est destiné à une structure de machine connexionniste travaillant en mode MIMD asynchrone, dans laquelle chaque processeur dispose de fonctions à usage général (general purpose processor).

Parmi les problèmes qui n'ont pas été abordés citons:

- la traduction d'un programme Prolog en un réseau sémantique étendu,
- la projection du graphe compilé sur une structure connexionniste de processeurs (allocation des processeurs),
- l'évaluation de l'efficacité de la compilation, de la méthode d'interprétation et du degré de parallélisme obtenu,
- la gestion de la mémoire locale de chaque processeur pour y implanter les variables nécessaires à l'algorithme.

Bibliographie

1. Deliyanni, A., and Kowalski, R.A. " Logic and Semantic Networks", Comm. ACM 22 3 (march 1979), pp. 184-192
2. Moldovan, D.I., and Yu-Wen Tung, SNAP "A VLSI Architecture for Artificial Intelligence Processing", Journal of Parallel and Distributed Computing 1985 2 pp 109-131.
3. Clark, K.L., and S. Gregory, "Parlog: Parallel Programming in Logic", ACM Trans. on Programming Languages, Vol. 8, No. 1, 1986.
4. Ueda, K., "Guarded Horn Clauses", ICOT technical report TR-103, 1985.
5. Shapiro, E., "A Subset of Concurrent Prolog and its Interpreter", ICOT technical report TR-003,1983.
6. Shapiro, E., "Concurrent Prolog: A Progress Report", IEEE Computer, august 1986, pp. 44-58.
7. Kahn, G., and MacQueen, D.B., "Coroutines and Networks of Parallel Processes", Information Processing 77: Proceedings of IFIP Congress 77, pp. 993-998.
8. Arvind and Brock, J.D., "Streams and Managers", Operating Systems Engineering, eds. M.Makegawa and L.A. Belady, Springer-Verlag, 1982, pp. 452-465, Lecture Notes in Computer Science, No. 143.
9. Hillis, D., "The Connection Machine", The MIT Press, 1985.
10. Kowalski, R.A. "Predicate Logic as Programming Language", Information Processing 74 : Proceedings of IFIP Congress 74, pp. 569-574.
11. Agha, G. "An Overview of Actor Languages", Sigplan Notices Vol 21, No 10, October 1986, pp 58-67.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique